Nicholas Chan
nipchan@ucsc.edu
5/27/2021
CSE13s Spring ‘21

Design Document:
Assignment 7: The Great Firewall of Santa Cruz

In assignment 7 we will be using bloom filters, which are represented by an array of bits like a bitvector. Words can be added to bloom filters and dictionaries which we will construct as a hashtable. Our bloom filter will be populated by reading in lines of badspeak.txt and oldspeak.txt with fscanf. More specifically, we call a hash function on the words from our input read in with fscanf while using different salts (given to us in the assignment pdf) We read in bad speak and old speak and set bits in the bloom filter.Based on whether the bloom filter is populated or not, we enter these inputted words into the hash table. Then once we get input from a user, we can check if any of the words from their input matches the words in the bloom filter, then if it is in the bloom filter we check if it is in the hashtable. If the word is in the hash table, we proceed by adding it to a list of words as either bad speak or old speak. The classification of either bad speak or old speak is cause for a correction for a word or words in input. We will be using regular expressions for parsing user entered data. Regular expressions will be used for determining valid words. If multiple words have the same key after hashing, we run into a hash collision. Hash collisions are handled by storing words that have the same key into a linked list, with the key now being directed towards the linked list.

Top Level:

Overview of Banhammer

- BloomFilter
- SPECK Cipher
    - Pass in a salt and a string that we want to be hashed and it will return a uint32_t
- Bitvector
    - All stuff the same from asgn5 without XOR. We need set, clr and get.
- HashTable
    - Usually has O(1) lookup and insert times
    - **Wrapper around an array linked lists**
        - We will be resolving hash collisions with chaining in linked lists
    - Things we can do with a hashtable
        - Create
        - Delete
        - Ask for size
        - Lookup string
        - Insert
        - Count non Null linked lists in hashtables
        - Print them out
- Linked Lists
    - Constructed of nodes
    - Create
    - Delete
    - Print (Supplied)_
    - Will be doubly linked with a 2 sentinel nodes to clean up logic
        - 1 head and 1 tail sentinel node
        - Values held in sentinels are typically irrelevant
- Lexical Analysis with Regular Expressions
    - We will be splitting input into different tokens or words with regular expressions (tokenizing or lexing)
    - Regular expression is just a pattern
- Parsing module (Given)
    - Next word
        - Scrapes input for words that match a regular expression
    - clear_words

Bloom Filter

Structure:
- Consists of 3 2 item uint64_t arrays for the primary, secondary and tertiary salt
- Bitvector of bloom filter

Insertion
- Create an index by calling hash function on the a salt and the input from oldspeak
  - Do this for each salt
  - Remember to keep the index within the bounds of the bloom filter (consider modulo)
- Call set_bit function to set bit at an index on the bloom filter bitvector
  - Do this for each salt

Usual functions
- Create
- Return size
- Delete

Hash Table

The hash table contains:
- Array of 2 uint64_t's for the hash table salt
- uint32_t for size
- bool for enabling or disabling the MTF rule
- An array of linked lists

ht_create takes in a uint32_t for size and bool for mtf and returns a HashTable pointer:
- Assigns the hash table salt, hash table size and hash table MTF status
- Allocates memory for an array of linked lists

ht_delete takes in a hash table pointer and deletes the hash table

ht_size returns the size of the hash table

ht_lookup searches for a node with some specified string through the hash table's array of linked lists and returns it. If no node is found, NULL should be returned

ht_insert takes in an oldspeak and newspeak string
- Create a hash key based on oldspeak
- If no linked list exists at the hash key index in the hash table, create a linked list at the index
  - Insert a linked list here

ht_count returns the number of indices in the hash table with linked list entries

Linked List

The linked list contains:

- 2 nodes
    - 1 head sentinel node
    - 1 tail sentinel node
- uint32_t for length
- bool for enabling or disabling the MTF rule

ll_create takes in a uint32_t for size and bool for mtf and returns a LinkedList pointer:

- Allocates memory for a linked list
- Creates nodes with NULLs for oldspeak and newspeak as the head and tail sentinel nodes
- Points the head and tail nodes at each other

ll_delete takes in a LinkedList pointer and deletes the linked list

ll_length returns the length of the linked list

ht_lookup searches for a node with some specified string through the linked list and returns it.

- If no node is found, NULL should be returned

ll_insert takes in an oldspeak and newspeak string

- If a node with the specified oldspeak exists in the linked list:
    - If MTF is enabled:
        - Move the node to the front of the linked list and return nothing
    - Return nothing
- If no node exists yet, create a node at the front of the linked list

Nodes

The linked list contains:

- 2 strings
    - 1 oldspeak string
    - 1 newspeak string
- 2 Node pointers
    - A pointer for the next node
    - A pointer for the previous node

ll_create takes in a uint32_t for size and bool for mtf and returns a LinkedList pointer:

- Allocates memory for a node as well as the strings taken in for oldspeak and newspeak
    - Allocating memory for strings is done with our own rendition of strdup
    - If no string is entered as input for oldspeak or newspeak, their respective pointers are assigned to NULL

- Pointers for next and prev are set to NULL

node_delete takes in a Node pointer and deletes the node
- This involves freeing the memory potentially allocated for oldspeak and newspeak

node_print takes in a Node pointer (node_print provided by pdf doc:
- Prints out the oldspeak and possibly newspeak strings possessed by the specified node

Bit Vector

Bitvector contains:
- uint32_t for length
- uint8_t array for the bit vector

bv_create takes in a uint32_t for size:
- Allocates memory for a BitVector

bv_delete takes in a pointer to a BitVector pointer and deletes it

bv_length takes in a BitVector pointer and returns its associated length

bv_clr_bit clears a bit at a specified index
- Bit clearing is setting a bit to 0
- This is done by performing an and operation on the complement of 1 and a BitVector where the only 0 in the complement of 1 is at the specified index

bv_get_bit takes in a BitVector pointer and uint32_t as an index:
- If a 1 is located in the bitvector at the index, return 1
    - Do this by performing an and operation with a 1 shifted by index number of bits and the bit vector
- Else return 0

bv_print takes in a BitVector pointer and prints the contents of the bit vector for debugging purposes

Design Process

I felt that the first thing I needed to understand for this assignment was its use of nodes, linked lists, hash tables and Bloom filters. I began by constructing the node ADT and testing to see if they were able to link to other nodes I created. Then I constructed linked lists out of the nodes I had. I then moved onto creating the Bloom filter which functioned pretty much the same as a bit vector with the addition of the functions found in the Speck module (hash function). Then I worked on the Hash table which was essentially an array of linked lists. I tested to make sure that all of these ADT's functioned properly before moving onto making my Banhammer module. The Banhammer module I created drew heavily from the given parsing module code as well as the example parser in the assignment 7 instructions pdf.

1. The first and simplest ADT I worked on was the node
    a. This node differed from its implementation in previous assignments as this would be used in a linked list and would be assigned strings to represent newspeak and oldspeak
    b. I used a strdup function that Professor Long provided on Piazza for allocating memory for my oldspeak and newspeak strings
2. The second ADT I worked on was the BitVector
    a. Not much had to be done for this as I had a working implementation from asgn5
    b. The Bitvector was used in the Bloom Filter
3. The third ADT I worked on was the Linked List, which was a logical move to me as the Node ADT could intuitively be connected to it
    a. The linked list was essentially a chain of nodes between two sentinel nodes representing the head and tail of the linked list.
        i. The head and tail sentinel nodes helped for establishing reference points for the move to front rule
            1. MTF would move any node that was looked up to the front, otherwise known as the position right behind the head
    b. Statistics like the total number of seeks and the total number of links traversed were recorded on the linked list module
4. The fourth ADT I worked on was the Hash Table which was essentially an array holding linked lists
    a. The possible number of linked lists able to be held in the Hash table was determined by the specified size that it was initialized with
    b. Like the Bloom filter, the hash table utilized the hash function from Speck.c to assign indices for linked list entries
    c. Many of the functions for the hash table were extensions off of the linked list functions

5. The last component of this assignment that I worked on was the banhammer.c module which utilized fscanf for reading in newspeak.txt and badspeak.txt to populate the Bloom filter and hash table
    a. As usual, I had getopt options for my command line options
    b. The parser.c module provided to me was helpful for reading and parsing input as words
        i. I used a regex expression: "[a-zA-Z0-9_]+((-|')?[a-zA-Z0-9_]+)*" to define what a word was
    c. Words parsed in as input were passed through my Bloom filter and then the hash table if necessary
    d. Words picked up as oldspeak or bad speak were assigned to either a badspeak linked list or oldspeak linked list
        i. Newspeak words or any other words were ignored
        ii. Depending on the sizes of the badspeak and oldspeak linked lists, the correct message from message.h would be printed out, along with the user's transgressions

I thought this assignment was pretty straight forward compared to asgn6. The use of nodes and linked lists seemed very useful to me as well as the notion of a hash table. I also found the idea of the Bloom filter as a screening device was fascinating. I don't think I would have ever thought of anything like that. I was also impressed by its functionality, despite its ability to only report ¾ths of an answer (inability to truly determine if a word is blacklisted). I was also surprised by how well the MTF rule actually improved performance. This is definitely a design choice that I will keep in mind for the future.

Side Notes:

- Bloomfilter operates off of a hash function.
- There will be 3 different hash function salts.
- A salt is additional data that we pass the data that we will hash. Take a string, hash it and it might return an integer value.
- We can use the same hash function, but if a different salt is added we get wildly different results.
- "Different salts make the hash come out differently, like flavoring a soup with different salts."
-
- Salts are 128 bit integers, but we don't have a native 128 bit int, so we will have to store salts in an array of 2 64 bit ints. Lower 64 bit and upper 64 bits will be stored.
- Bloom filter checks on input words
- If you use a word that is not in the bloom filter, -> not a bad word and fine to use

- If you use a word that is in the bloom filter, -> we check the hash table -> if word does not contain a translation, it is a bad speak word
- (see page 11 for more specifics on possible outcomes)